Under consideration for publication in J. Functional Programming

Calculating a Linear-Time Solution to the Densest-Segment Problem

SHARON CURTIS
Oxford University, UK
and SHIN-CHENG MU
Academia Sinica, Taiwan
(*e-mail:* sharon.curtis@ndph.ox.ac.uk, scm@iis.sinica.edu.tw)

Abstract

The problem of finding a densest segment of a list is similar to the well-known maximum segment sum problem, but its solution is surprisingly challenging. We give a general specification of such problems, and formally develop a linear-time online solution, using a sliding-window style algorithm. The development highlights some elegant properties of densities, involving partitions that are decreasing and all right-skew.

1 Introduction

Processing large datasets, such as those containing DNA sequences, requires efficient algorithms. The data analysis problem we examine here involves finding a *segment* (contiguous subsequence) of a list of elements with maximum *density* (to be defined in the next section). While this problem resembles the well-known maximum segment sum problem (MSS), a standard textbook example of program derivation (Kaldewaij, 1990), the internal structure of the segment density problem is more intricate, and finding a linear-time solution is much harder.

Our aim in this paper is the derivation of a linear-time functional program to solve a generalised segment density problem, in an elegant and clear way. We present two instances of such density problems: MMS (maximum mean segment) and MDS (maximum density segment), which has applications to analysis of DNA sequences. The MMS problem we believe is new; MDS has been solved before (Chung & Lu, 2004; Goldwasser *et al.*, 2005), but in an imperative setting, where it is difficult to see the structure and correctness of the algorithms amongst the details.

Our algorithm derivation uses a traditional program calculation approach, at least initially. However, as several of the proofs are tricky or sizable, for additional reassurance the theorems and their proofs are also coded into the dependently-typed language/theorem prover Agda (Norell, 2007) and the AOPA library (Mu *et al.*, 2009); the complete proof is around 3500 lines of Agda code.

In this paper, we outline the main proofs; the rest of the proofs can be found in a supplement containing full details, available online (Curtis & Mu, 2014), along with a Haskell implementation and Agda code. The paper is structured as follows: in Section 2 we give a description and specification of the densest segment problem, along with the two instances MMS and MDS. The development of the algorithm then proceeds in two stages: in Section 3 we develop the structure of the algorithm, which processes the input list using a *sliding-window* technique. The second stage is presented in Section 4, which explores properties of the window to develop functions sufficient to allow the algorithm to run in linear time. The resulting datatypes for the window, and the algorithm's implementation, are presented in Section 5 along with an analysis of its performance. We conclude and summarise related work in Section 6.

2 The Densest Segment Problem

Segment problems have been well-examined in the literature, especially in the late 1980s and early 1990s, when their study formed part of a wider body of work on constructive algorithmics. Some of this calculational work resulted in imperative-style implementations, e.g. (Kaldewaij, 1990; Van Den Eijnde, 1990; Rem, 1988), while others used a functional style for their development, e.g. (Bird, 1987; Jeuring, 1993; Swierstra & de Moor, 1993)

The segment problems studied were optimisation problems on a list of input elements, where it was desired to find a segment that was optimal with regards to some function. Such a function might evaluate a segment on its length (i.e. wanting a shortest or longest segment, such as in Jeuring & Meertens (1989)), or in other more interesting ways, such as trying to find the largest rectangle under a histogram (Van Den Eijnde, 1990). Often the segments of interest would also have to satisfy some predicate, and depending on the properties of the predicate, it might be possible to find a linear-time algorithm to solve the problem, often in a "sliding-window" style (Zantema, 1992).

The general densest segment problem we address in this paper describes a particular specialised class of segment problems, using optimisation functions that satisfy a *density property* (see Section 2.1) and a predicate broadly related to segment length. In this rest of this section we give a description of this class of segment problems, before presenting two specific instances: MDS (maximum density segment), which features in the existing literature, and MMS (maximum mean segment), the latter of which we believe is novel.

2.1 Description

The problem's input data is a sequence of elements of type *Elem*. Each segment (contiguous subsequence) of the input elements has a *width*,

$$\mathit{width} :: [\mathit{Elem}] \to \mathbb{R}^{\geq 0},$$

with the property that an empty list has zero width, and adding more elements at either end increases width. Formally:

$$\mathit{width}\ [\,] = 0, \tag{1}$$

$$\mathit{width}\ x < \mathit{width}\ (x +\!\!+\, y) > \mathit{width}\ y, \tag{2}$$

where x and y are any non-empty lists of elements. For example, the function *length* is such a width function, but there are other more interesting possibilities (see the MDS problem in Section 2.2 below).

Not all segments are considered as candidate solutions to the problem, and there is a constraint on segment width: a lower bound given by $L : \mathbb{R}^{>0}$. (Without such a constraint, the solution is often trivial.) This predicate checks whether a segment is wide enough:

$$\begin{aligned} &\mathit{wide} && :: \; [\mathit{Elem}] \to \mathit{Bool} \\ &\mathit{wide}\; x && = \; \mathit{width}\; x \geq L. \end{aligned}$$

Evaluation of segments occurs by means of a *density* function

$$d \; :: [\mathit{Elem}] \to \mathbb{R}.$$

Informally, the density of a segment can be thought of as comparing the "weight" of a segment's elements to its width, in some way. (For specific examples, please see the functions d_{MMS} and d_{MDS} in Sections 2.3 and 2.2 below.) Comparison between segments is carried out using the relation

$$\begin{aligned} &(\leq_d) && :: [\mathit{Elem}] \to [\mathit{Elem}] \to \mathit{Bool} \\ &x \leq_d y && \Leftrightarrow \; d\, x \leq d\, y. \end{aligned}$$

As a shorthand, we will write $x <_d y$ for $x \leq_d y \wedge y \not\leq_d x$, and write $\geq_d$ and $>_d$ for the converses of $\leq_d$ and $<_d$.

However, the function d is not strictly needed in the calculation and implementation of a solution: what matters is that there exists a relation $\leq_d$ for comparing segments that is reflexive, transitive and total on non-empty lists (anti-symmetry is not required, as two different segments may be equally dense). It is also required that the relation $\leq_d$ satisfies the following *density property*: for any non-empty lists $x, y :: [\mathit{Elem}]$,

$$x \;\leq_d\; x \mathbin{+\!\!+} y \quad \Leftrightarrow \quad x \leq_d y \quad \Leftrightarrow \quad x \mathbin{+\!\!+} y \;\leq_d\; y. \tag{3}$$

The above must also hold if $\leq_d$ is replaced with $<_d$, $\geq_d$ or $>_d$.

In summary, the densest segment problem is the finding of a maximally-dense segment with respect to a relation $\leq_d$ that satisfies the density property, subject to a lower bound L on width.

2.2 Maximum Density Segment (MDS)

In the MDS problem, each element has a (strictly positive) *size*:

$$\mathit{size} \; :: \; \mathit{Elem} \to \mathbb{R}^{>0},$$

and the width of a segment is the sum of the sizes of its elements:

$$\begin{aligned} &\mathit{width}_{\mathit{MDS}} && :: \; [\mathit{Elem}] \to \mathbb{R}^{\geq 0} \\ &\mathit{width}_{\mathit{MDS}} && = \; \mathit{sum} \cdot \mathit{map}\; \mathit{size}. \end{aligned}$$

Note that $\mathit{width}_{\mathit{MDS}}$ satisfies the width properties (1) and (2) stated above.

Also, each element in the MDS problem has a *weight*:

$$\mathit{weight}_{\mathit{MDS}} \; :: \; \mathit{Elem} \to \mathbb{R},$$

and the density of a non-empty segment is calculated by dividing its total weight by its width:

$$\begin{aligned} d_{MDS} &:: [Elem] \to \mathbb{R} \\ d_{MDS}\ seg &= (sum\ (map\ weight_{MDS}\ seg))/(width_{MDS}\ seg). \end{aligned}$$

A straightforward arithmetical calculation shows that $\leq_{d_{MDS}}$ does indeed satisfy the density property (3).

For the MDS problem, densities can be visualised accurately: when each element is depicted as a rectangle as wide as its size, with area corresponding to its weight, the density of a segment is simply its average height, e.g. see Figure 1. Thus the MDS problem is to find a maximally dense segment with respect to d_{MDS}, subject to a lower bound L on width. For example, the densest segment of the list of elements in Fig. 1, with width at least 9, is the segment $[(14,7),\ (20,4)]$, which has density $34/11 \approx 3.1$.

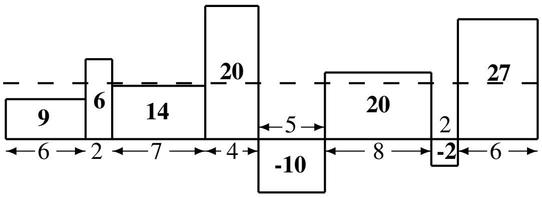

Fig. 1. A list of elements, with weights in bold and sizes as indicated. The density d_{MDS} of the whole list is $(9+6+14+20-10+20-2+27)/(6+2+7+4+5+8+2+6) = 84/40 = 2.1$, which is shown by the height of the dashed line.

The MDS problem has applications in bioinformatics, to the analysis of genetic material to find fragments with high densities of certain characteristics. For example, sections of DNA or RNA that are *GC-rich* – with a high density of bases that are guanine (G) or cytosine (C) – are of interest because they are likely to indicate portions containing genes (Han & Zhao, 2009). Another example is that sections of DNA that have a high density of mutations are of interest because they can provide clues pointing to the three-dimensional molecular structure.

The solution of the MDS problem has a varied history: in Huang (1994), a simpler version of the MDS problem was considered, where all elements have unit size. Huang noticed that with a lower bound L on segment width, there is a maximally dense segment no wider than $2L-1$, which leads to a simple $O(nL)$ algorithm, for n input elements. Later, the complexity of a MDS solution was improved to $O(n \log L)$ (Lin *et al.*, 2002). Goldwasser *et al.* (2002) studied a variation of MDS that also used an upper bound U on segment width, and they presented an $O(n)$ algorithm for the simpler case of MDS when all elements have unit size (also an instance of MMS), and an $O(n \log(U-L+1))$ algorithm for elements with variable sizes. Another published algorithm claimed to be linear, but wasn't; details are recorded in Chung & Lu (2004). Then both Chung & Lu (2003; 2004) and Goldwasser *et al.* (2005) produced similar algorithms to take linear time to solve MDS with lower and upper bounds.

2.3 Maximum Mean Segment (MMS)

In the MMS problem, the function to measure the width of a segment is not specified further than it having to obey the width properties (1) and (2).

$$width_{MMS} :: [Elem] \rightarrow \mathbb{R}^{\geq 0}$$

The density of a non-empty segment, for the MMS problem, is the mean weight of its elements:

$$\begin{aligned} d_{MMS} &:: [Elem] \rightarrow \mathbb{R} \\ d_{MMS}\ seg &= (sum\ (map\ weight\ seg))/(length\ seg). \end{aligned}$$

A straightforward arithmetical calculation shows that $\leq_{d_{MMS}}$ satisfies the density property (3). Thus the MMS problem is to find a maximally dense segment with respect to d_{MMS}, subject to a lower bound L on segment width.

One application of the MMS problem is the analysis of chronological data. For example, consider a blood glucose measurement history from an individual with diabetes (a disorder of blood sugar control): an answer to the question "When were blood sugar levels worst over the past year?" might prove useful in identifying factors impairing control. Typical data consists of a sequence of timestamped blood glucose readings in chronological order, such as that illustrated in Figure 2.

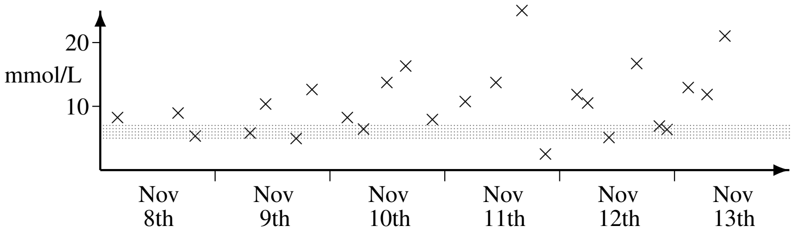

Fig. 2. A sequence of blood glucose readings, where the dotted area indicates the target range. Usually, several months' worth of data would be available.

This situation is an instance of the MMS problem: each element is a blood glucose reading, and its weight is how far it deviates from the target blood sugar range. Thus the density of a segment is how far its readings are outside the target range, on average, over the timespan of that segment. Here it makes sense to put a lower bound on that timespan (for example, a week) as spurious extreme readings can happen occasionally, so it is not the single worst reading that is sought, but an average over a suitably-long period.

We expect that MMS problems can be identified in many other application areas, including analysis of other kinds of medical history data.

3 Development

3.1 Specification & Refinement

In this section, we continue on from the datatypes and definitions that have already been given in Section 2.1, to arrive at a specification for the densest segment problem.

One standard way to produce all the segments of a list is to take all prefixes of suffixes:

$$
\begin{array}{lll}
\mathit{prefixes}, \mathit{suffixes}, \mathit{segments} & :: & [a] \to [[a]], \\
\mathit{prefixes} & = & \mathit{inits} \\
 & & \textbf{where}\ \mathit{inits}\ [\,] = [[\,]] \\
 & & \qquad \mathit{inits}\ (x : xs) = [[\,]] \mathbin{+\!\!+} \mathit{map}\ (x :)\ (\mathit{inits}\ xs), \\
\mathit{suffixes} & = & \mathit{tails} \\
 & & \textbf{where}\ \mathit{tails}\ [\,] = [[\,]] \\
 & & \qquad \mathit{tails}\ (x : xs) = (x : xs) : \mathit{tails}\ xs, \\
\mathit{segments} & = & \mathit{concat} \cdot \mathit{map}\ \mathit{prefixes} \cdot \mathit{suffixes}.
\end{array}
$$

Later, we will write $x \sqsubseteq y$ to denote that x is a prefix of or equal to y, and use $\sqsubset$ for the strict version.

To select maximal segments, we will use a binary operator $\upharpoonleft_{\unlhd}$, which selects the maximum of two arguments with respect to a total ordering $\unlhd$:

$$
\begin{array}{rcll}
x \upharpoonleft_{\unlhd} y & = & x, & \text{if } y \unlhd x\,; \\
 & = & y, & \text{otherwise.}
\end{array}
$$

In the case of a tie between the two arguments, $\upharpoonleft_{\unlhd}$ selects the left-hand argument. Later, we will also need an operator $\upharpoonright_{\unlhd}$, which is defined similarly to break ties in favour of the right-hand argument. We will abbreviate $\upharpoonleft_{\leq_d}$ and $\upharpoonright_{\leq_d}$ by $\upharpoonleft_d$ and $\upharpoonright_d$ respectively, to avoid cumbersome subscripts.

Given a binary operator $\oplus$ (such as $\upharpoonleft_{\unlhd}$ or $\upharpoonright_{\unlhd}$) that selects the maximum of two values, the following function selects the maximum of a list:

$$
\begin{array}{lcl}
\mathit{max}_{\oplus}\ [x] & = & x \\
\mathit{max}_{\oplus}\ (x : xs) & = & x \oplus (\mathit{max}_{\oplus}\ xs).
\end{array}
$$

We can now specify the problem of finding a densest segment as:

$$
\mathit{max}_{\upharpoonleft_d} \cdot \mathit{filter}\ \mathit{wide} \cdot \mathit{segments}. \tag{4}
$$

At this stage, $\upharpoonright_d$ could just as easily have been used, and the choice of $\upharpoonleft_d$ will be justified later in Section 3.4 (just after the proof of Lemma 3.5).

The above specification (4) can be a starting point for a calculation to derive a solution. However, during the subsequent development, a number of issues occur, which we find easier to manage if the initial specification is refined first.

The first issue is that the above specification does not describe a total function, because it may be that no segments of the input are wide enough. Unfortunately this lack of totality forms a major obstacle to deriving an algorithm to solve this problem, as will be discussed later in Section 3.4.

Attempting a derivation using relational algebra rather than functional programming calculus, for example in the style described in Bird & de Moor (1997), does not help, as the requirement for totality remains. To ensure totality, there needs to be unified treatment of wide segments and those that are not wide enough. Using a *Maybe* type is one possibility, but this leads to a very fiddly derivation.

Instead, we use another way to model treating segments differently depending on whether they are wide enough, by generalising $\leq_d$ to a relation $\preceq$ with the following properties:

$$
\begin{aligned}
& \preceq \text{ is reflexive, transitive, and total;} && (5)\\
\neg \mathit{wide}\ x \wedge \mathit{wide}\ y \quad &\Rightarrow \quad x \prec y, && (6)\\
\mathit{wide}\ x \wedge \mathit{wide}\ y \quad &\Rightarrow \quad (x \preceq y \Leftrightarrow x \leq_d y), && (7)\\
\neg \mathit{wide}\ x \wedge \neg \mathit{wide}\ y \quad &\Rightarrow \quad (x \preceq y \Leftrightarrow \mathit{width}\ x \leq \mathit{width}\ y), && (8)
\end{aligned}
$$

where $x \prec y$ is a shorthand for $x \preceq y \wedge y \npreceq x$. Informally, the requirement (5) ensures that taking a maximum from a non-empty list of segments is always possible, (6) says that wide segments are always strictly better than those not wide enough, (7) says that amongst wide lists, a denser segment is better, and (8) says that amongst not-wide-enough segments, wider is better (the reason for which will appear as a consequence of the algorithmic development, at the end of Section 3.4). Note also that the definition of $\preceq$ ensures that density comparisons are only made when the segments are both wide enough and therefore non-empty.

In this way, selecting a densest segment can be carried out by $\mathit{max}_{\upharpoonleft_{\preceq}}$, which we will abbreviate to $\mathit{max}_{\preceq}$. Our new version of the specification is then

$$
\mathit{mds} \quad = \quad \mathit{max}_{\preceq} \cdot \mathit{segments}. \qquad (9)
$$

This is a total specification, unlike (4). It produces a slightly different output, which can be used to solve the original problem: if the resulting segment is not at least L wide, then there were no wide segments. Otherwise, the output is a densest segment with respect to $\leq_d$, which will be the leftmost such segment, as $\upharpoonleft_{\preceq}$ is used.

3.2 Initial Calculation Steps

Starting from the *mds* specification, we take a standard formal development route to calculate for the cases $[\,]$ and $(a:x)$. After obtaining $\mathit{mds}\ [\,] = [\,]$, we then proceed as follows:

$$
\begin{aligned}
& \mathit{mds}\ (a:x)\\
= \quad & \quad \{\text{definition of } \mathit{mds}\ (9)\}\\
& (\mathit{max}_{\preceq} \cdot \mathit{segments})\ (a:x)\\
= \quad & \quad \{\text{definitions of } \mathit{segments} \text{ and } \mathit{suffixes}\}\\
& \mathit{max}_{\preceq}\ (\mathit{concat}\ (\mathit{map\ prefixes}\ ((a:x) : \mathit{suffixes}\ x)))\\
= \quad & \quad \{\text{definitions of } \mathit{map} \text{ and } \mathit{concat}\}\\
& \mathit{max}_{\preceq}\ (\mathit{prefixes}\ (a:x) +\!\!+\ \mathit{concat}\ (\mathit{map\ prefixes}\ (\mathit{suffixes}\ x)))\\
= \quad & \quad \{\mathit{max} \text{ distributes over } +\!\!+, \text{ since } \upharpoonleft_{\preceq} \text{ is associative}\}\\
& \mathit{max}_{\preceq}\ (\mathit{prefixes}\ (a:x))\ \upharpoonleft_{\preceq}\ \mathit{max}_{\preceq}\ (\mathit{concat}\ (\mathit{map\ prefixes}\ (\mathit{suffixes}\ x)))\\
= \quad & \quad \{\text{definitions of } \mathit{segments} \text{ and } \mathit{mds}\}\\
& \mathit{max}_{\preceq}\ (\mathit{prefixes}\ (a:x))\ \upharpoonleft_{\preceq}\ \mathit{mds}\ x\\
= \quad & \quad \{\text{definition, see below}\}\\
& \mathit{mdp}\ (a:x)\ \upharpoonleft_{\preceq}\ \mathit{mds}\ x.
\end{aligned}
$$

Above, we used this definition of a maximally-dense prefix (*mdp*) with respect to $\preceq$:

$$mdp \;=\; max_{\preceq} \cdot prefixes. \tag{10}$$

Maximally-dense prefixes will feature heavily in the coming calculations, so we now pause the development in order to collect some basic properties of *mdp* that we will need later.

3.3 Properties of mdp

Firstly, as can be deduced from its definition, *mdp* returns a prefix of its input for any list of elements x:

$$mdp\ x \;\sqsubseteq\; x, \tag{11}$$

Secondly, this lemma describes the relationship between *mdp* and segment width:

Lemma 3.1
For any list of elements x,

$$\begin{aligned} width\,x \leq L \quad &\Rightarrow \quad x = mdp\,x, &(12)\\ wide\,x \quad &\Leftrightarrow \quad wide\,(mdp\,x). &(13)\end{aligned}$$

The implication (12) expresses that taking the *mdp* of a narrow segment returns the segment itself, and (13) expresses that taking the *mdp* of a segment does not affect whether it meets the width constraint or not.

Proof
The implication (12) is proved using a straightforward induction on x. A short calculation yields $wide\ x\ \Rightarrow\ wide\ (mdp\ x)$, which together with (12) is sufficient to show (13). (For details, see the supplement.) □

The function *mdp* has two reasonably straightforward-to-prove properties, which show that extending a segment can only increase the length and density of the densest prefix:

Lemma 3.2 (*Monotonicity of mdp*)
For any lists of elements x, y such that $x \sqsubseteq y$, we have that

$$\begin{aligned} mdp\,x \quad &\sqsubseteq \quad mdp\,y, &(14)\\ mdp\,x \quad &\preceq \quad mdp\,y. &(15)\end{aligned}$$

The next lemma says that a prefix shorter than the *mdp* of a segment is not maximally dense. This relies on *mdp* selecting the leftmost amongst maximally-dense prefixes:

Lemma 3.3
For any lists of elements x, y such that $x \sqsubset y$, we have that

$$x \sqsubset mdp\,y \quad \Rightarrow \quad x \prec mdp\,y$$

Finally, we have the following lemma that says if x is "sandwiched" in between *mdp* y and y, then the maximally-dense prefixes of x and y are the same:

Lemma 3.4 (*Sandwich Lemma*)
For any lists of elements x and y,

$$mdp\,y \sqsubseteq x \sqsubseteq y \;\Rightarrow\; mdp\,x = mdp\,y. \tag{16}$$

3.4 A sliding-window algorithm

Returning to the algorithmic development, in Section 3.2 we calculated the following alternative definition for *mds*, which requires finding a densest prefix for all suffixes of the input:

$$\begin{aligned} \mathit{mds}\ [\,] &= [\,] \\ \mathit{mds}\ (a:x) &= \mathit{mdp}\ (a:x)\ \uparrow_{\preceq}\ \mathit{mds}\,x. \end{aligned}$$

Despite being tail-recursive, this function is inefficient, as computing *mdp* for every suffix of the input would take too long. If, however, *mdp* itself can be expressed as a *foldr*, then the problem can be solved by performing the paired computation $\langle \mathit{mds}, \mathit{mdp}\rangle$[1], which can be carried out as a *foldr*: this would compute an optimal segment ($\mathit{mds}\ x$) and an optimal prefix ($\mathit{mdp}\ x$) for each suffix x of the input. Then the value $\mathit{mdp}\ (a:x)$ would be readily available at each step, when calculating $\mathit{mds}\ (a:x)$ from $\mathit{mds}\,x$.

[A technical aside: afficionados of the morphism zoo may like to note that the $\langle \mathit{mds}, \mathit{mdp}\rangle$ computation illustrates the *zygomorphism* computational pattern (Malcolm, 1990), where the value of one expression is computed from a fold that pairs it with a second expression that is directly computable from a fold (“zygo” means “yoked”). The zygomorphic nature of our algorithm is why we are using total functions (as discussed in Section 3.1): zygomorphisms exist in the category of sets and total functions (**Fun**), but are not guaranteed to exist in the categories of partial functions (**PFun**) and relations (**Rel**).]

Unfortunately, *mdp* is not a *foldr*. If it were, then $\mathit{mdp}\ (a:x)$ could be calculated from a and $\mathit{mdp}\,x$ alone, but consider the list $[(1,5),(9,2),(8,2)]$ of $(\mathit{weight},\mathit{size})$ elements for the MDS problem, with a segment width lower bound $L = 1.8$. Here $\mathit{mdp}\ [(9,2),(8,2)] = [(9,2)]$, but $\mathit{mdp}\ [(1,5),(9,2),(8,2)] = [(1,5),(9,2),(8,2)]$, which is not computable solely from $(1,5)$ and $[(9,2)]$.

One of the surprising features of the *mds* problem, however, is that *mdp* not being a *foldr* does not seem to matter. In the above example, $\mathit{mdp}\ [(9,2),(8,2)]$ is denser than $\mathit{mdp}\ [(1,5),(9,2),(8,2)]$, and thus the latter cannot be a solution to the whole problem. This suggests that perhaps $\mathit{mdp}\ (a:x)$ is not needed when x has a denser shorter prefix. This is the case, and we will prove a slightly generalised version: if $\mathit{mdp}\ (u \mathbin{+\!\!+} x)$ is longer than $u \mathbin{+\!\!+} \mathit{mdp}\ x$, then $\mathit{mdp}\ (u \mathbin{+\!\!+} x)$ is less dense than $\mathit{mdp}\ x$. The following lemma formalises this property:

Lemma 3.5 (*Overlap Lemma*)
For all lists of elements u and x, we have

$$u \mathbin{+\!\!+} \mathit{mdp}\ x \sqsubset \mathit{mdp}\ (u \mathbin{+\!\!+} x) \quad\Rightarrow\quad \mathit{mdp}\ (u \mathbin{+\!\!+} x) \prec \mathit{mdp}\ x.$$

Proof
We do a case analysis on the width of x.

Case $\neg \mathit{wide}\,x$.
By (12) we have $x = \mathit{mdp}\,x$. The assumption thus simplifies to $u \mathbin{+\!\!+} x \sqsubset \mathit{mdp}\ (u \mathbin{+\!\!+} x)$ which is false, as $\mathit{mdp}\ (u \mathbin{+\!\!+} x)$ is a prefix of $u \mathbin{+\!\!+} x$, from property (11).

[1] The “split” operator is defined by $\langle f, g\rangle\, a = (f\,a, g\,a)$.

Case *wide x*.
As from (2), prepending only increases width, then by (13) we know that $wide\ (mdp\ x)$ and $wide\ (mdp\ (u \mathbin{+\!\!+} x))$, and thus from the definition of $\prec$ and $\preceq$ (7), the goal simplifies to proving that $mdp\ (u \mathbin{+\!\!+} x) <_d mdp\ x$.

Note that since $u \mathbin{+\!\!+} mdp\ x \sqsubset mdp\ (u \mathbin{+\!\!+} x)$, we have $mdp\ (u \mathbin{+\!\!+} x) \ = \ u \mathbin{+\!\!+} mdp\, x \mathbin{+\!\!+} z$, for some non-empty z. We first prove that $u \mathbin{+\!\!+} mdp\, x \mathbin{+\!\!+} z \ <_d \ z$:

$$
\begin{array}{ll}
 & u \mathbin{+\!\!+} mdp\, x \ \sqsubset \ mdp\,(u \mathbin{+\!\!+} x) \\
\Rightarrow & \{ \ \text{lemma 3.3} \ \} \\
 & u \mathbin{+\!\!+} mdp\, x \ \prec \ mdp\,(u \mathbin{+\!\!+} x) \\
\Leftrightarrow & \{ \ \text{decomposition of } mdp\ (u \mathbin{+\!\!+} x)\text{, as above} \ \} \\
 & u \mathbin{+\!\!+} mdp\, x \ \prec \ u \mathbin{+\!\!+} mdp\, x \mathbin{+\!\!+} z \\
\Rightarrow & \{ \ \text{definition of } \preceq \text{ (7), as } wide\,(mdp\ x) \text{ and width increases with } \mathbin{+\!\!+} \text{ (2)} \ \} \\
 & u \mathbin{+\!\!+} mdp\, x \ <_d \ u \mathbin{+\!\!+} mdp\, x \mathbin{+\!\!+} z \\
\Rightarrow & \{ \ \text{density property (3)} \ \} \\
 & u \mathbin{+\!\!+} mdp\, x \mathbin{+\!\!+} z \ <_d \ z.
\end{array}
$$

Then we prove that $z \leq_d mdp\, x$:

$$
\begin{array}{ll}
 & u \mathbin{+\!\!+} mdp\, x \mathbin{+\!\!+} z \ = \ mdp\,(u \mathbin{+\!\!+} x) \\
\Rightarrow & \{ \ \text{prefix property of } mdp \text{ (11)} \} \\
 & u \mathbin{+\!\!+} mdp\, x \mathbin{+\!\!+} z \ \sqsubseteq \ u \mathbin{+\!\!+} x \\
\Leftrightarrow & \{ \ \text{property of prefixes} \ \} \\
 & mdp\, x \mathbin{+\!\!+} z \ \sqsubseteq \ x \\
\Rightarrow & \{ \ mdp\, x \text{ maximum wrt } \preceq) \ \} \\
 & mdp\, x \mathbin{+\!\!+} z \ \preceq \ mdp\, x \\
\Rightarrow & \{ \ \text{definition of } \preceq \text{ (7), as } wide\,(mdp\ x) \text{ and width increases with } \mathbin{+\!\!+} \text{ (2)} \ \} \\
 & mdp\, x \mathbin{+\!\!+} z \ \leq_d \ mdp\, x \\
\Rightarrow & \{ \ \text{density property (3), with } \geq_d \ \} \\
 & z \ \leq_d \ mdp\, x.
\end{array}
$$

Thus by transitivity we conclude that $u \mathbin{+\!\!+} mdp\, x \mathbin{+\!\!+} z \ <_d \ mdp\, x$, which is the same as $mdp\ (u \mathbin{+\!\!+} x) \ <_d \ mdp\, x$. □

The Overlap Lemma (3.5) suggests an alternative to computing *mdp x* for each suffix *x* of the input list of elements: each new element *a* is added to the front of the prefix produced from the previous stage of the computation, and then *mdp* of this segment is computed.

[An aside: this strategy is what suggests the use of $\upharpoonleft_{\preceq}$ in the definition (10) of *mdp*: in the case that there is more than one densest prefix of a segment, $max_{\upharpoonleft_{\preceq}}$ chooses the shortest, thus leaving fewer elements to be processed later on than if the $\upharpoonright_{\preceq}$ operator were to be used. In turn, this justifies the use of $\upharpoonleft_{\preceq}$ in the definition (9) of *mds* as well.]

Formally, this computation can be expressed as the following function *wp*:

$$\begin{aligned} wp\ [\,] &= [\,] \\ wp\ (a : x) &= mdp\ (a : wp\ x). \end{aligned} \tag{17}$$

(The name *wp* is intended to abbreviate “window processing”.) Note that unlike *mdp*, the function *wp* is clearly a *foldr*. We now hope to use the value of *wp* instead of *mdp* for each suffix, in order to compute *mds*. That is, given the following function *ms*:

$$\begin{aligned} ms\ [\,] &= [\,] \\ ms\ (a : x) &= wp\ (a : x)\ \uparrow_{\preceq}\ ms\ x, \end{aligned} \tag{18}$$

we need to show that $mds = ms$, which occurs in Section 3.6.

The benefit of using *ms* may not be obvious. Recall that in Section 3.4, we were unable to use the paired computation $\langle mds, mdp \rangle$ to compute *mds* efficiently because *mdp* is not a *foldr*. Now, however, as *wp* is a *foldr*, we can use a paired computation $\langle ms, wp \rangle$, expressible as a *foldr* that makes one call to *mdp* in each step. Letting $mswp = \langle ms, wp \rangle$, a standard *tupling* transformation gives us:

$$\begin{aligned} mswp\ [\,] &= ([\,],[\,]) \\ mswp\ (a : x) &= (w' \uparrow_{\preceq} m, w') \\ &\qquad \textbf{where}\ (m, w) = mswp\ x \\ &\qquad\qquad\quad w' = mdp\ (a : w). \end{aligned}$$

This follows the paradigm known as a *sliding-window* algorithm (Zantema, 1992): such a computation maintains a segment of the input sequence (the *window*), that “slides” along the sequence at each step of the algorithm, always in the same direction. Above, w is the window storing the value $wp\ x$ from the previous step of the computation, and the application of $mdp \cdot (a :)$ produces an updated window w' that is a prefix of $a : w$, thus sliding the window to the left. (The name *wp* stands for “window processing”.)

[Another aside: this is the point where it is clear how $\preceq$ compares segments that are not wide enough, as expressed earlier in (8). When the algorithm has barely started, and is still examining suffixes of the input that are not wide enough, it is not clear what *wp* should return, but for the first suffix x that is wide enough, $mds\ x$ will be x itself, and therefore *wp* will need to retain all the elements of x. This means that on segments that are not wide enough, $\preceq$ should favour wider segments.]

Given suitable list structures to enable constant time computation of $\uparrow_{\preceq}$, the above algorithm can be implemented in linear time, provided that the computation of $mdp \cdot (a :)$ can be done in amortised constant time. This is our goal in Section 4: choosing suitably efficient data structures for the window.

3.5 Properties of wp *and* ms

As *mdp* returns a prefix of its input (11), a short inductive proof starting from the definition of *wp* (17) shows that $wp\,x$ also returns prefixes:

$$wp\,x \sqsubseteq x, \tag{19}$$

for any sequence of elements x.

The following lemma shows that *wp* and *ms* have similar relationships to the width of segments that *mdp* does (see Lemma 3.1).

Lemma 3.6
For any list of elements x,

$$width\,x \leq L \quad \Rightarrow \quad x = wp\;x = mds\;x, \tag{20}$$

$$wide\,x \quad \Leftrightarrow \quad wide\,(wp\;x) \quad \Leftrightarrow \quad wide\,(mds\;x). \tag{21}$$

3.6 Correctness of the sliding-window algorithm

Now we carry out our task to prove that $mds\;x = ms\;x$.

Proof
We use an induction on x, and the $[\,]$ case is immediate from the definitions of *mds* and *ms*. For the $a : x$ case, the proof obligation (after standard use of the definition of *wp* and the induction hypothesis that $mds\;x = ms\;x$) is

$$mdp\;(a : x) \uparrow_{\preceq} mds\;x \;=\; mdp\;(a : wp\,x) \uparrow_{\preceq} mds\;x, \tag{22}$$

Noting how $\preceq$ is defined, we start from a case analysis on the width of x.

Case $\neg wide\,x$. By (20), $x = wp\,x$, and thus (22) holds.

Otherwise, we split the case for $wide\,x$ into two cases, based on whether the left-hand side of (22) equates to $mdp\;(a : x)$ or $mds\;x$. As the operator $\uparrow_{\preceq}$ selects its left-hand argument when comparing two segments that are equal with respect to $\preceq$, the two cases are $mdp\,(a : x) \prec mds\,x$ and $mdp\,(a : x) \succeq mds\,x$. As $wide\,x$ holds, these simplify, using the definition of $\preceq$, to the cases $mdp\,(a : x) <_d mds\,x$ and $mdp\,(a : x) \geq_d mds\,x$ respectively.

Case $wide\,x$ and $mdp\,(a : x) <_d mds\,x$.
As the left-hand side of (22) reduces to $mds\;x$, we need to show that the right-hand side also reduces to $mds\,x$. This necessitates proving that $mdp\;(a : wp\,x) \prec mds\;x$:

$$\begin{array}{ll}
 & mdp\;(a : wp\,x) \prec mds\;x \\
\Leftarrow & \{\ \text{since } mdp\,(a : x) <_d mds\,x\ \} \\
 & mdp\;(a : wp\,x) \preceq mdp\;(a : x) \\
\Leftarrow & \{\ \text{monotonicity of } mdp \text{ as in (15)}\ \} \\
 & a : wp\,x \sqsubseteq a : x \\
\Leftarrow & \{\ wp \text{ produces prefixes (19)}\ \} \\
 & true.
\end{array}$$

Case $wide\,x$ and $mdp\,(a : x) \geq_d mds\,x$.
This is the difficult case, and clearly if we can establish that

$$mdp\,(a : x) \;\geq_d\; mds\,x \quad \Rightarrow \quad mdp\,(a : x) = mdp\;(a : wp\,x), \tag{23}$$

in the circumstance that $wide\,x$, then (22) will hold.

To establish (23) we need a separate result, Lemma 3.7 below, which generalises the prepending $(a\,:)$ operation in (23) to the prepending of a list $(z \mathbin{+\!\!+})$. To use the lemma, we let $z := [a]$ and $y := x$ in (24) of the lemma, and then the assumption required, on the left-hand side of (24), is $\mathit{mds}\,x \preceq \mathit{mdp}\,(a : x)$. This can be obtained by using the facts that $\mathit{mdp}\,(a : x) \geq_d \mathit{mds}\,x$ and $\mathit{wide}\,x$, together with (13) and (21). □

Lemma 3.7
For any finite lists of elements y and z,

$$\mathit{mds}\,y \;\preceq\; \mathit{mdp}\,(z \mathbin{+\!\!+} y) \;\;\Rightarrow\;\; \mathit{mdp}\,(z \mathbin{+\!\!+} y) = \mathit{mdp}\;(z \mathbin{+\!\!+} \mathit{wp}\,y). \tag{24}$$

Proof
The proof proceeds by induction on y. The case for $y := [\,]$ is routine. For the case $a : y$, the assumption is

$$\mathit{mds}\,(a : y) \preceq \mathit{mdp}\,(z \mathbin{+\!\!+} (a : y)),$$

and for the induction hypothesis, we can assume that

$$\mathit{mds}\;y \preceq \mathit{mdp}\,(v \mathbin{+\!\!+} y) \;\;\Rightarrow\;\; \mathit{mdp}\,(v \mathbin{+\!\!+} y) = \mathit{mdp}\;(v \mathbin{+\!\!+} \mathit{wp}\,y),$$

for any finite list of elements v.

We need to show that $\mathit{mdp}\,(z \mathbin{+\!\!+} a : y) = \mathit{mdp}\;(z \mathbin{+\!\!+} \mathit{wp}\,(a : y))$. One crucial idea is to use the sandwich lemma to turn the proof obligation from an equality to a prefix relation, which then allows us to use the overlap lemma, the key property that guarantees that we need not consider prefixes that are too long. The proof goes as follows:

$$\mathit{mdp}\,(z \mathbin{+\!\!+} a : y) = \mathit{mdp}\;(z \mathbin{+\!\!+} \mathit{wp}\,(a : y))$$

$\Leftarrow$ { sandwich lemma (16) }

$$\mathit{mdp}\,(z \mathbin{+\!\!+} a : y) \sqsubseteq z \mathbin{+\!\!+} \mathit{wp}\,(a : y) \sqsubseteq z \mathbin{+\!\!+} a : y$$

$\Leftrightarrow$ { by (19), $\mathit{wp}\,(a : y) \sqsubseteq a : y$ }

$$\mathit{mdp}\,(z \mathbin{+\!\!+} a : y) \sqsubseteq z \mathbin{+\!\!+} \mathit{wp}\,(a : y)$$

$\Leftarrow$ { induction, with $v = z \mathbin{+\!\!+} [a]$, writing P for $\mathit{mds}\,y \preceq \mathit{mdp}\,(z \mathbin{+\!\!+} a : y)$ }

$$\mathit{mdp}\,(z \mathbin{+\!\!+} a : \mathit{wp}\,y) \sqsubseteq z \mathbin{+\!\!+} \mathit{wp}\,(a : y) \;\wedge\; P$$

$\Leftrightarrow$ { definition of wp }

$$\mathit{mdp}\,(z \mathbin{+\!\!+} a : \mathit{wp}\,y) \sqsubseteq z \mathbin{+\!\!+} \mathit{mdp}\,(a : \mathit{wp}\,y) \;\wedge\; P$$

$\Leftarrow$ { overlap lemma 3.5, contra-positive form (see note below) }

$$\mathit{mdp}\,(a : \mathit{wp}\,y) \preceq \mathit{mdp}\,(z \mathbin{+\!\!+} a : \mathit{wp}\,y) \;\wedge\; P$$

$\Leftarrow$ { induction }

$$\mathit{mdp}\,(a : \mathit{wp}\,y) \preceq \mathit{mdp}\,(z \mathbin{+\!\!+} a : y) \;\wedge\; P$$

$\Leftarrow$ { by monotonicity (15), and $a : \mathit{wp}\,y \sqsubseteq a : y$ }

$$\mathit{mdp}\,(a : y) \preceq \mathit{mdp}\,(z \mathbin{+\!\!+} a : y) \;\wedge\; P$$

$\Leftrightarrow$ { expanding P }

$$\mathit{mdp}\,(a : y) \preceq \mathit{mdp}\,(z \mathbin{+\!\!+} a : y) \;\wedge\; \mathit{mds}\,y \preceq \mathit{mdp}\,(z \mathbin{+\!\!+} a : y)$$

$\Leftrightarrow$ { maximum }

$$
\begin{array}{ll}
 & \mathit{mdp}\,(a:y) \uparrow_{\preceq} \mathit{mds}\,y \;\preceq\; \mathit{mdp}\,(z \mathbin{+\!\!+} a:y) \\
\Leftrightarrow & \quad \{ \text{ definition of } \mathit{mds} \; \} \\
 & \mathit{mds}\,(a:y) \preceq \mathit{mdp}\,(z \mathbin{+\!\!+} a:y) \\
\Leftarrow & \quad \{ \text{ assumption } \} \\
 & \mathit{true}.
\end{array}
$$

Note that the step above using the Overlap Lemma relies on the lists $\mathit{mdp}\,(z \mathbin{+\!\!+} a : \mathit{wp}\,y)$ and $z \mathbin{+\!\!+} \mathit{mdp}\,(a : \mathit{wp}\,y)$ both being prefixes of $z \mathbin{+\!\!+} a : \mathit{wp}\,y$, in order to simplify the $\not\sqsubseteq$ in $\mathit{mdp}\,(z \mathbin{+\!\!+} a : \mathit{wp}\,y) \not\sqsubseteq z \mathbin{+\!\!+} \mathit{mdp}\,(a : \mathit{wp}\,y)$ to $\sqsupset$. □

4 Window

Recall the definition of *wp* (17), which describes the sliding of the window in the main algorithm:

$$
\begin{array}{lcl}
\mathit{wp}\,[\,] & = & [\,] \\
\mathit{wp}\,(a:x) & = & \mathit{mdp}\,(a:\mathit{wp}\,x)
\end{array}
$$

Our goal is to choose a suitable data structure for the window so that the computation of $\mathit{mdp} \cdot (a\,:)$ takes amortised constant time, thus allowing the overall algorithm to be linear.

In this section, we will examine the properties of *mdp* in order to see how best to represent the window, so that suitable data structures for the window can be chosen in Section 5. In what follows, let the window *wp x* at the previous step of the computation be denoted by w, and let $w' = a : w$.

4.1 Window Header

Recall that the function *mdp* takes the best prefix of w' with respect to $\uparrow_{\preceq}$. In the case that $\neg \mathit{wide}\ w'$, then we just have $\mathit{mdp}\ w' \;=\; w'$, from Lemma 3.1. In the case when $\mathit{wide}\ w'$, another straightforward step can be taken, given part (6) of the definition of $\preceq$, which says thatwide segments are always better than too-short segments. Thus when $\mathit{wide}\ w'$, it is also the case that $\mathit{wide}\ (\mathit{mdp}\ w')$, as formalised in Lemma 3.1, (13). This means that when $\mathit{wide}\ w'$, the prefix $\mathit{mdp}\ w'$ must include sufficiently many elements that it is wide enough; these elements can be thought of as a "header" prefix of w'.

Let the function *hsplit* be such that it splits a segment into its header prefix, and the rest of its elements.

$$\mathit{hsplit} \;::\; [\mathit{Elem}] \to ([\mathit{Elem}],[\mathit{Elem}]),$$

A possible definition of *hsplit* is given later (see Figure 8 in Section 5.2), but for now, we just need the following property: if $\mathit{hsplit}\ w' = (h,x)$, we have $h \mathbin{+\!\!+} x = w'$, where h is the header prefix, so that either

- $\mathit{width}\ h < L$ and $x = [\,]$, or
- $\mathit{width}\ (\mathit{init}\ h) < L \leq \mathit{width}\ h$ (i.e. h is the shortest wide prefix), where $\mathit{init}\ (y \mathbin{+\!\!+} [b]) \;=\; y$.

Furthermore, recall that part (7) of the definition of $\preceq$ says that the better of two wide segments is determined by $\leq_d$. Thus when $\mathit{wide}\ w'$, taking a maximum prefix using the

$\upharpoonleft_{\preceq}$ operator in *mdp* w' is the same as $max_{\upharpoonleft_d}$ (*prefixes* w'). This means that for wide w', we can restrict our attention to prefixes of w' of the form $h +\!\!+ y$, where $(h,x) = hsplit\ w'$ and $y \sqsubseteq x$, and compare densities with respect to $\upharpoonleft_d$.

We thus represent a window as a pair, where the first component is the header h,

$$\textbf{type}\ Window = ([Elem], \ldots),$$

and the second component is some structure yet to be determined, to represent the rest of the window. From this structure we will need to find the segment that produces the densest prefix of the whole window with respect to $\upharpoonleft_d$, when prepended by the header h.

4.2 Densest Prefixes

Let $(h,x) = hsplit\ w'$; our goal is to find a prefix of w' of the form $h +\!\!+ y$, that is densest with respect to $\upharpoonleft_d$. One possibility is that $y = [\,]$ and h itself is the densest such segment; otherwise, there must be a non-empty y such that $h <_d h +\!\!+ y$, which is equivalent to $h <_d y$, by the density property (3). This leads us to investigate non-empty prefixes of x that are denser than h; indeed perhaps choosing y to be a maximally-dense prefix of x might result in a densest prefix $h +\!\!+ y$?

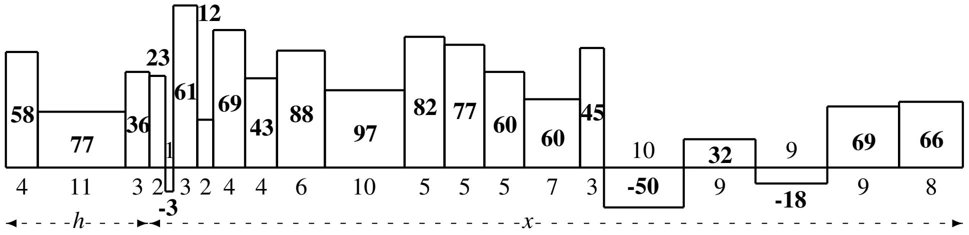

Fig. 3. A running MDS example: the list of elements [(58,4), (77,11), (36,3), (23,2), (-3,1), (61,3), (12,2), (69,4), (43,4), (88,6), (97,10), (82,5), (77,5), (60,5), (60,7), (45,3), (-50,10), (32,9), (-18,9), (69,9), (66,8)], with weights indicated in bold. Lower bound $L = 16$; the compulsory header is h.

Unfortunately, this is not the case: there may be a prefix of x that is slightly less dense than y, but wider, and that produces a denser prefix overall when prepended by the header h. A concrete example is illustrated in Figure 3, where a maximally-dense non-empty prefix of x is $y = [(23,2), (-3,1), (61,3), (12,2), (69,4)]$, with density 13.5, resulting in a density of 11.1 for $h +\!\!+ y$. However, extending y on the right by the following elements $z = [(43,4), (88,6), (97,10), (82,5), (77,5), (60,5)]$ yields an improved density of 12 for $h +\!\!+ y +\!\!+ z$.

This is disappointing, but maybe a densest prefix can still be of use? Although the example above illustrates that y (a maximally-dense prefix of x) can be bettered by a prefix wider (longer) than y, perhaps non-empty prefixes shorter than y can be ruled out? It turns out that this is indeed the case, which will be shown in what follows.

4.3 Right-Skew Segments

First, we will need to examine some properties of densest prefixes. Note that if a non-empty list of elements $y = y' +\!\!+ y''$ is a maximally-dense prefix of x, then as y' is also a prefix of x, we have $y' \leq_d y$. Then, from the density property, this is equivalent to $y' \leq_d y''$. This leads us to the following definition:

Definition 4.1 (*Right-Skew*)
A non-empty list z of elements is called *right-skew* if for all n such that $0 < n < \mathit{length}\ z$, we have $\mathit{take}\ n\ z \leq_d \mathit{drop}\ n\ z$. Let us denote this by the predicate

$$\mathit{rightskew}\ (\leq_d)\ ::\ \mathit{Elem} \rightarrow \mathit{Bool}.$$

The idea of right-skew segments originated with Lin *et al.* (2002); informally, a segment is right-skew when chopping it into two always results in a right-hand side that is at least as dense than the left. However, note that density does not increase in a monotonic fashion: although shorter prefixes of right-skew segments are often less dense than longer prefixes, this is merely a trend, and should not be relied upon. For an example of a right-skew list that illustrates this lack of monotonicity, see Figure 4.

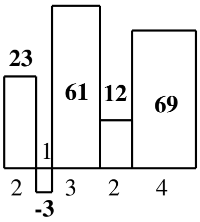

Fig. 4. The segment [(23,2), (-3,1), (61,3), (12,2), (69,4)] is right-skew.

From this reasoning, we have that every densest prefix of a list of elements is right-skew (the converse does not hold):

Lemma 4.2
Let y be a non-empty prefix of a list of elements x, such that y is maximally dense with respect to $\leq_d$. Then y is right-skew.

One property of right-skew segments we will need is that the concatenation of two right-skew segments in the correct order is itself right-skew:

Lemma 4.3
If z and z' are both right-skew and $z \leq_d z'$, then $z +\!\!+ z'$ is right-skew.

Returning to the development, recall that given a non-empty list of elements y that is a maximally dense prefix of x, we wish to rule out the use of prefixes shorter than y. The following lemma allows us to do so:

Lemma 4.4
Let z_1, z_2, z_3 be non-empty list of elements such that $z_2 \leq_d z_3$. Then

$$z_1 \uparrow_d (z_1 +\!\!+ z_2) \uparrow_d (z_1 +\!\!+ z_2 +\!\!+ z_3)\ =\ z_1 \uparrow_d (z_1 +\!\!+ z_2 +\!\!+ z_3).$$

In words, this means that the densest among the three lists z_1, $z_1 +\!\!+ z_2$ and $z_1 +\!\!+ z_2 +\!\!+ z_3$ with respect to $\uparrow_d$ is either z_1 or $z_1 +\!\!+ z_2 +\!\!+ z_3$. Thus, as a densest prefix y of x is right-skew, and hence any proper division of y into $y = y' +\!\!+ y''$ results in $y' \leq_d y''$, Lemma 4.4 allows us to deduce that of the three lists h, $h \mathbin{+\!\!+} y'$ and $h \mathbin{+\!\!+} y$, the densest with respect to $\uparrow_d$ is not $h \mathbin{+\!\!+} y'$. Thus non-empty prefixes of y can be eliminated from consideration. This can be shown formally, as an application of the following theorem:

Theorem 4.5
Let z and y be non-empty lists of elements with y right-skew. Then

$$max_{\uparrow_d}\ (map\ (z\mathbin{+\!\!+})\ (prefixes\ y)) \ = \ z \uparrow_d (z \mathbin{+\!\!+} y). \tag{25}$$

This helps suggest a strategy for structuring the rest of the window, as follows. Having shown that the interior of any right-skew prefix y of x can be eliminated from consideration, it seems reasonable that choosing the longest possible right-skew prefix of x (let us denote this prefix by y_0) may help reduce the remaining computation within the rest of the window. Furthermore, having established h and $h \mathbin{+\!\!+} y_0$ as possible candidates for the *mdp* of the window, but with no need to consider prefixes in between, it seems reasonable to try the same step with the rest of the window. That is, let y_1 be the longest right-skew prefix of $x \setminus\!\setminus y_0$ (here, the list subtraction operator is denoted by $\setminus\!\setminus$, so that $x \setminus\!\setminus y_0$ contains the elements of x apart from its initial prefix of y_0). Then applying Theorem 4.5 with $z = h \mathbin{+\!\!+} y_0$ and $y = y_1$ removes prefixes between $h \mathbin{+\!\!+} y_0$ and $h \mathbin{+\!\!+} y_0 \mathbin{+\!\!+} y_1$ from consideration.

Similarly, let y_2 be the longest right-skew prefix of $x \setminus\!\setminus (y_0 \mathbin{+\!\!+} y_1)$, and so on, resulting in a partition of x into right-skew lists of elements $[y_0, y_1, \ldots y_k]$. Repeated application of Theorem 4.5 leads us to the following theorem:

Theorem 4.6
Given $h : [Elem]$ and $xs :: [[Elem]]$ such that each list in xs is right-skew, we have

$$max_{\uparrow_d}\,(map\,(h\mathbin{+\!\!+})\,(prefixes\,(concat\,xs))) \ = max_{\uparrow_d}\,(map\,((h\mathbin{+\!\!+}) \cdot concat)\,(prefixes\,xs)).$$

This theorem says that to compute $max_{\uparrow_d}\,(h \mathbin{+\!\!+} y_0 \mathbin{+\!\!+} y_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} y_k)$ we only need to consider the ends of each partition: h, $h \mathbin{+\!\!+} y_0$, $h \mathbin{+\!\!+} y_0 \mathbin{+\!\!+} y_1$, etc.

Later on, we will need an alternative version of the above theorem, as follows:

Corollary 4.7
Given $h : [Elem]$ and $xs :: [[Elem]]$ such that each list in xs is right-skew, we have

$$max_{\uparrow_h}\,(prefixes\,(concat\ xs)) \ = \ concat\,(max_{\uparrow_{h\mathbin{+\!\!+}}}\,(prefixes\ xs)),$$

where $\uparrow_h$ and $\uparrow_{h\mathbin{+\!\!+}}$ abbreviate $\uparrow_{\leq_d^h}$ and $\uparrow_{\leq_d^{h\mathbin{+\!\!+}}}$ respectively, where

$$\begin{aligned} x \leq_d^h y &\quad\equiv\quad h \mathbin{+\!\!+} x \leq_d h \mathbin{+\!\!+} y, \\ xs \leq_d^{h\mathbin{+\!\!+}} ys &\quad\equiv\quad h \mathbin{+\!\!+} concat\ xs \leq_d h \mathbin{+\!\!+} concat\ ys. \end{aligned}$$

This partitioning of x into right-skew lists will be how the rest of the window is structured, but first we need to formally examine the properties of the partition we have just created, and prove some properties we will need.

4.4 Decreasing Right-Skew Partitions

In this section, we present the concept of *decreasing right-skew partitions*, which were proposed by Lin *et al.* (2002). We also investigate the properties of these partitions, and show that they are the same as the partitions proposed above for the window structure.

A decreasing right-skew partition (abbreviated DRSP) of a list of elements is a partition of the list into segments, such that each segment is right-skew and the densities of the segments are strictly decreasing from left to right. Formally:

Definition 4.8 (*DRSP*)
Let *xs* be a partition of a list of elements *x*, that is, where $concat\ xs\ =\ x$. The list *xs* is a *decreasing right-skew partition (DRSP)* of *x* when

- $all\ (rightskew\ (\leq_d))\ xs$, and
- $sdec\ (\leq_d)\ xs$,

where $sdec\ (\trianglelefteq)$ is the predicate that expresses that a list is strictly decreasing with respect to a linear ordering $\trianglelefteq$, i.e. $sdec\ (\trianglelefteq)\ [x_1 \ldots x_n]$ holds precisely when $x_1 \triangleright x_2 \triangleright \ldots \triangleright x_n$.

Note that the right-skew property ensures that each segment of the partition is non-empty. An example of a DRSP can be seen in Figure 5.

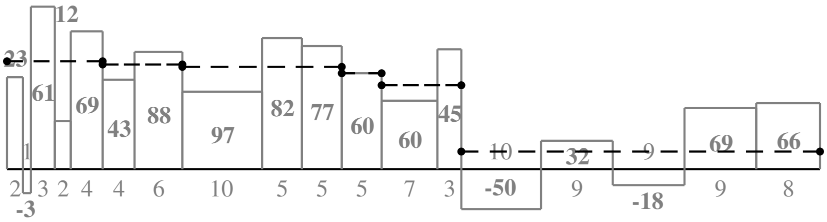

Fig. 5. The (unique) DRSP of the list x from Fig. 3 is $[[(23,2),(-3,1),(61,3),(12,2),(69,4)],$ $[(43,4),(88,6)],$ $[(97,10),(82,5),(77,5)],$ $[(60,5)],$ $[(60,7),(45,3)],$ $[(-50,10),(32,9),(-18,9),$ $(69,9),$ $(66,8)]]$. The densities of its segments are 13.5, 13.1, 12.8, 12, 10.5 and 2.2 respectively, indicated by the dashed lines representing the average heights of the segments.

The first important property that we need is that DRSPs are unique:

Theorem 4.9 (*DRSP uniqueness*)
Let *x* be a list of elements. There exists precisely one decreasing right-skew partition of *x*.

This is a surprising property of such partitions, and indeed we did not find the proof of uniqueness in Lin *et al.* (2002) convincing, so have provided our own in the supplement to this paper.

A consequence of DRSP uniqueness that we will need is the necessary existence of a function $drsp_{\leq_d} : [Elem] \rightarrow [[Elem]]$ that produces the unique DRSP, given a list of elements; a definition of such a function will be given later (see just after Theorem 4.12).

The following property notes that a DRSP has a “rotational symmetry”: to visualise this, turn Figure 5 upside-down, and it still depicts a DRSP:

Theorem 4.10 (*DRSP rotation*)
For any non-empty list of elements *x*,

$$drsp_{\leq_d}\ x\ =\ reverse \cdot map\ reverse \cdot drsp_{\geq_d}\ (reverse\ x).$$

The DRSP rotation property is not needed for this algorithmic development, but it is included here for the record.

The final property of DRSPs concerns how they can be constructed. We will need some preliminaries. Firstly, the longest right-skew prefix of a list of elements is also the longest densest prefix of the list:

Lemma 4.11
Let y be a non-empty list of elements. Then the longest right-skew prefix of y is also the longest of the maximally-dense prefixes of y.

It is the this lemma that illustrates how the partition of the right-hand side of the window $x = concat\ [y_0, y_1, \ldots, y_k]$, as suggested in the previous section, results in a DRSP. Recall that the segment y_0 is selected as the longest right-skew prefix of x, and therefore, by Lemma 4.11, it must also be the longest densest prefix of x. This ensures that the next segment y_1 must be of strictly lower density than y_0 (otherwise $y_0 +\!\!+\, y_1$ would be at least as dense as y_0, contradicting y_0 being the longest of the maximally dense prefixes of x). Thus $y_0 >_d y_1$, and repeatedly selecting longest right-skew segments results in a partition of right-skew segments of decreasing densities.

Lemma 4.11 describes the first of several possible methods of constructing decreasing right-skew partitions:

Theorem 4.12 (*DRSP construction*)
The DRSP of a list of elements can be constructed by any of the following methods:

(i) repeatedly taking longest maximum-density prefixes
(ii) repeatedly taking longest right-skew prefixes[2]
(iii) repeatedly taking longest minimum-density suffixes[3]
(iv) repeatedly taking longest right-skew suffixes

The following function illustrates one of the above ways (i) to build a DRSP:

```
drsp1    :: [Elem] → [[Elem]]
drsp1 [] = []
drsp1 x  = y : drsp1 (drop (length y) x)
             where y = max↾d (prefixes x).
```

However, for the representation of the window in our algorithm, we are not usually going to be constructing a complete DRSP from a list of elements; the window will already consist of a header and the DRSP of the remaining elements, and we will want to incrementally update the partition as new elements are added.

The following lemma allows a DRSP to be updated from the left-hand side:

Lemma 4.13

[2] A longest right-skew prefixes approach is used in Goldwasser *et al.* (2005) and Lin *et al.* (2002).

[3] The algorithm in Chung & Lu (2004) slides the window from left to right, using a mirror-image DRSP structure, the construction of which is based on taking longest minimum-density prefixes.

Let z be a right-skew list of elements, and ys a DRSP. Then $prepend\ z\ ys$ is a DRSP, where

$$\begin{aligned}
&prepend && :: \ [Elem] \to [[Elem]] \to [[Elem]] \\
&prepend\ z\ [\,] && = \ [z] \\
&prepend\ z\ (y:ys) && = \ \textbf{if}\ z \leq_d y\ \textbf{then}\ prepend\ (z +\!\!+\, y)\ ys \\
& && \ \ \textbf{else}\ z:y:ys.
\end{aligned}$$

Above, the *prepend* function repeatedly joins the list of elements z with the leftmost segment of ys (justified by Lemma 4.3), until the densities are once more decreasing.

This leads to the following alternative way to build a DRSP, using a function *addl* that appends one element to the left of an existing DRSP:

$$\begin{aligned}
&drsp && :: \ [Elem] \to [[Elem]] \\
&drsp && = \ foldr\ addl\ [\,], \\[1ex]
&addl && :: \ Elem \to [[Elem]] \to [[Elem]] \\
&addl\ a\ xs && = \ prepend\ [a]\ xs.
\end{aligned}$$

Note that the right-skew property that *prepend* requires of its first argument applies, as any singleton list $[a]$ is right-skew. The function *addl* will be useful in our final code.

4.5 Use of the DRSP

To summarise where we have reached in the development: given a list of elements $w' = a : w$ for which we want to calculate $mdp\ w'$, we split the window w' using $hsplit\ w' = (h,x)$, and represent the list x by its DRSP. The data structure for the window is therefore:

$$\textbf{type}\ Window \ = \ ([Elem],[[Elem]]),$$

where the first component is the compulsory header h, and the second is the rest of the window, partitioned into $drsp\ x$. Furthermore, Theorem 4.6 established that we don't need to consider prefixes that end in the interior of right-skew segments in the partition of x.

Note that the lists in the *Window* datatype should be considered as abstract representations at this stage, as we have not yet finalised whether to use cons-lists, snoc-lists, or some other queue representation, as befits a sliding-window algorithm. We will make these decisions in Section 5.2, to allow the final algorithm to run in linear time.

Having chosen a data structure for the window, we will need some representation-changing functions to convert between a list of elements and its partition into a DRSP:

$$\begin{aligned}
&wbuild && :: \ [Elem] \to Window \\
&wbuild && = \ (id \times drsp) \cdot hsplit, \\[1ex]
&wflatten && :: \ Window \to [Elem] \\
&wflatten\ (h,xs) && = \ h +\!\!+\, concat\ xs.
\end{aligned}$$

Above, the function *wbuild* constructs a window from a list of elements[4], while *wflatten* does the opposite, flattening a window back to a list. Thus we have $wflatten \cdot wbuild = id$.

[4] The "product functor" $(\times)$ is defined by $(f \times g)\ (a,b) = (f\,a, g\,b)$.

To work on a more abstract level, we define a function *wcons*, so that *wcons a* adds an element a to the left side of the window. It can be seen as $(a\,:)$ lifted to the *Window* datatype, although it performs much more work — repartitioning the header, and updating the DRSP:

$$\begin{array}{lcl} \mathit{wcons} & :: & \mathit{Elem} \to \mathit{Window} \to \mathit{Window} \\ \mathit{wcons}\ a\ (h,\mathit{xs}) & = & (h', \mathit{foldr\ addl\ xs}\ x) \\ & & \quad \textbf{where}\ (h',x)\ =\ \mathit{hsplit}\ (a:h). \end{array}$$

We have made progress: the data structure for the window has been chosen, we can construct and update a DRSP by adding elements incrementally to the front of it. Now we just need the *mdp* of the window.

4.6 Extracting the Densest Prefix

To find the densest prefix with respect to $\rceil_d$, we need to make use of Theorem 4.6, which guarantees that for a partition of the elements beyond the compulsory header into right-skew segments (such as the DRSP we are using), only the positions of the input corresponding to the ends of these segments need to be considered.

Furthermore, the decreasing densities of the DRSP will enable us to pinpoint exactly where the *mdp* of the window elements is to be found, as follows. Empirical examination suggests that the densities of the prefixes h, then $h +\!\!+\, y_0$ and so on up to $h +\!\!+\, y_0 \ldots +\!\!+\, y_k$ form a simple hill shape, illustrated by our running example in Figure 6. To be precise: the hill for the window prefix densities of a header & DRSP is bitonic, consisting of a strictly increasing ascent on the left, followed by a (non-strictly) decreasing descent. Either side of the hill can be empty, for example it might just be a gentle decreasing slope. For the moment, please assume that our assertion about this hill shape is true, and this will soon be addressed formally in Lemma 4.14.

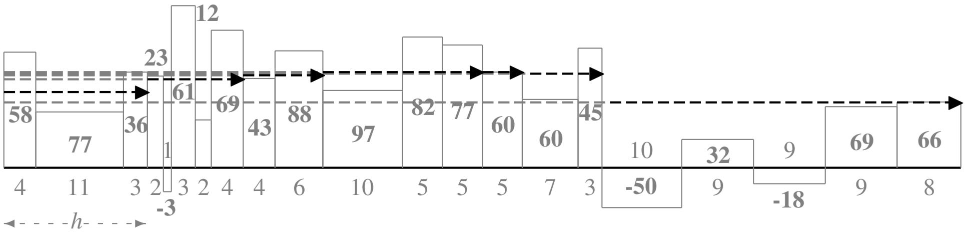

Fig. 6. In our running example, the options for prepending the header h onto prefixes given by the DRSP results in prefix densities (from left to right) 9.5 11.1, 11.6, 12, 12, 11.8 and 8.2. This bitonic hill shape can be seen outlined in the arrowheads.

From this hill shape, it is now easy to identify the *mdp*: the densest window prefix(es) can be found at the top of the hill, and in particular, the densest prefix wrt $\rceil_d$ (which is what *mdp* uses) will be at the top of the hill, at the left-hand end if it turns out that the top of the hill is a plateau (i.e. more than one prefix having maximal density).

Finding the top of the hill can be achieved by a simple traverse from either the left or right side of the hill; however it will be seen later that starting from the right will have a crucial effect on the overall efficiency of the algorithm, enabling each element to be processed once only (more details are given in Section 5.2). Here is the definition of a function that, given a header h and a DRSP, chops off segments of the DRSP from the right-hand side, stopping at the left of the top of the hill:

$$\begin{array}{lcl} \mathit{maxchop} & :: & [\mathit{Elem}] \rightarrow [[\mathit{Elem}]] \rightarrow [[\mathit{Elem}]] \\ \mathit{maxchop}\ h\ [\,] & = & [\,] \\ \mathit{maxchop}\ h\ (\mathit{xs} \mathbin{+\!\!+} [x]) & = & \textbf{if}\ h \mathbin{+\!\!+} \mathit{concat}\ \mathit{xs} <_d x\ \textbf{then}\ \mathit{xs} \mathbin{+\!\!+} [x] \\ & & \textbf{else}\ \mathit{maxchop}\ h\ \mathit{xs}. \end{array}$$

The *maxchop* function repeatedly removes a segment x when $h \mathbin{+\!\!+} \mathit{concat}\ \mathit{xs} \not<_d x$, which from the density property is equivalent to $h \mathbin{+\!\!+} \mathit{concat}\ \mathit{xs} \geq_d h \mathbin{+\!\!+} \mathit{concat}\ (\mathit{xs} \mathbin{+\!\!+} [x])$. In other words, *maxchop* chops from the right whilst going up the right-hand side of the hill, only stopping if it runs out of hill (the $[\,]$ case) or if it finds a strict decrease in density, as the given condition $h \mathbin{+\!\!+} \mathit{concat}\ \mathit{xs} <_d x$ is equivalent to $h \mathbin{+\!\!+} \mathit{concat}\ \mathit{xs} <_d h \mathbin{+\!\!+} \mathit{concat}\ (\mathit{xs} \mathbin{+\!\!+} [x])$, from the density property. To illustrate, *maxchop* applied to our running MDS example in Figure 6 returns the list of segments $[[(23,2),(-3,1),(61,3),(12,2),(69,4)],[(43,4),(88,6)],$ $[(97,10),(82,5),(77,5)]]$.

The following lemma shows that the list of segments that *maxchop* returns is strictly ascending in density when prepended by the header h, and thus shows formally that we really do have a simple bitonic hill shape as described at the beginning of this section.

Lemma 4.14
Let h be a non-empty list of elements, and $\mathit{xs} \mathbin{+\!\!+} [x]$ be a list of non-empty segments with strictly *decreasing* densities. If $h \mathbin{+\!\!+} \mathit{concat}\ \mathit{xs} <_d x$, then the following list of segments has strictly *increasing* densities:

$$\mathit{map}\ ((h\mathbin{+\!\!+}) \cdot \mathit{concat})\ (\mathit{prefixes}\ (\mathit{xs} \mathbin{+\!\!+} [x])).$$

We can then use this lemma to show that *maxchop* does indeed compute the *mdp* for us, which is what we require:

Theorem 4.15
Let h a non-empty list of elements, and let *xs* be a (possibly empty) list of non-empty segments having strictly decreasing densities. Then

$$h \mathbin{+\!\!+} \mathit{concat}\ (\mathit{maxchop}\ h\ \mathit{xs})\ =\ \mathit{max}_{\uparrow_d}\ (\mathit{map}\ ((h\mathbin{+\!\!+}) \cdot \mathit{concat})\ (\mathit{prefixes}\ \mathit{xs})).$$

Later, in our proofs, we will actually use the following corollary, which is the same property stated differently:

Corollary 4.16
Let h a non-empty list of elements, and let *xs* be a (possibly empty) list of non-empty segments having strictly decreasing densities. Then

$$\mathit{maxchop}\ h\ \mathit{xs}\ =\ \mathit{max}_{\uparrow_{h+\!+}}\ (\mathit{prefixes}\ \mathit{xs}).$$

($\uparrow_{h+\!+}$ is defined in Corollary 4.7.)

For convenience, we also define a wrapper function for *maxchop* that operates on the whole window:

$$\begin{array}{lcl} \mathit{wmaxchop} & :: & \mathit{Window} \rightarrow \mathit{Window} \\ \mathit{wmaxchop}\ (h,\mathit{xs}) & = & (h, \mathit{maxchop}\ h\ \mathit{xs}). \end{array}$$

5 Putting Everything Together

Let us remind ourselves of the development so far, the outline of which is summarised in Figure 7.

The refined specification (9):

$mds \ = \ max_{\preceq} \cdot segments$

We proved that

$mds \ = \ ms$

Computing *ms* is obtained from

$\langle ms, wp \rangle$,

which can be expressed as a *foldr*.

Recap of definitions:

$ms \ [\,] \ = \ [\,]$

$ms \ (a : x) \ = \ wp \ (a : x) \ \uparrow_{\preceq} \ ms \ x$

$wp \ [\,] \ = \ [\,]$

$wp \ (a : x) \ = \ mdp \ (a : wp \ x)$

$mdp \ = \ max_{\preceq} \cdot prefixes$

Fig. 7. Summary of the algorithmic development so far.

In Section 2 the original specification was refined to produce the function *mds*, then in Section 3, a sliding-window algorithm *ms* was shown to satisfy that specification, where the value of *ms* is obtained from the paired computation $\langle ms, wp \rangle$. Section 4 was devoted to finding a faster way to compute $mdp \ \cdot \ (a :)$ in the inductive step for *wp*, the window-processing function, and as a result we obtained a datatype *Window* to use for representing *wp x*.

In this section we will put the results of Sections 3 and 4 together to calculate our algorithm to solve the density problem. We will then refine the data structure in Section 5.2 to allow efficient implementation of certain operations. For now, using a standard list data structure allows us to see what is going on in the algorithm more easily.

5.1 Structuring the Window

Our algorithm for obtaining a maximally dense segment is $ms \ = \ fst \ \cdot \ \langle ms, wp \rangle$, from which we can introduce the window structure as follows:

$$\begin{array}{cl} & fst \ \cdot \ \langle ms, wp \rangle \\ = & \quad \{ \text{ window identity, see page 20 } \} \\ & fst \ \cdot \ \langle ms, wflatten \cdot wbuild \cdot wp \rangle \\ = & \quad \{ \text{ identity, composition } \} \\ & fst \ \cdot \ (id \times \ wflatten) \ \cdot \ \langle ms, wbuild \cdot wp \rangle \\ = & \quad \{ \text{ projections } \} \\ & fst \ \cdot \ \langle ms, wbuild \cdot wp \rangle . \end{array}$$

Our task is now to push *wbuild* into the calculation of *wp* so that all the window processing is done with the *Window* datatype. Thus we define

$$\begin{aligned} mwp &:: [Elem] \to ([Elem], Window) \\ mwp &= \langle ms, wbuild \cdot wp \rangle, \end{aligned}$$

and we will aim to calculate an inductive definition of *mwp*.

The base case is straightforward: $mwp\ [\,] \ = \ ([\,],([\,],[\,]))$. For the inductive case we calculate:

$$\begin{aligned} & mwp\ (a : x) \\ = \quad & \{ \text{ definitions of } ms \text{ and } wp \ \} \\ & (mdp\ (a : wp\ x) \uparrow_{\preceq} ms\ x,\ wbuild\ (mdp\ (a : wp\ x))) \\ = \quad & \{ \text{ since } wflatten \cdot wbuild = id \ \} \\ & (wflatten\ u \uparrow_{\preceq} ms\ x,\ u) \\ & \textbf{where } u = wbuild\ (mdp\ (a : wp\ x)). \end{aligned}$$

Consider the above binding $u = wbuild\ (mdp\ (a : wp\ x))$. If we can somehow manage to push *wbuild* to the right and show that $u = f\ (wbuild\ (wp\ x))$ for some f, we will then have an inductive definition for *mwp*. To do this, since $wbuild = (id \times drsp) \cdot hsplit$, it will be helpful to know how *hsplit* interacts with $(a\,:)$ and *mdp*. Firstly, the following lemma shows how *hsplit* and $(a\,:)$ exchange:

Lemma 5.1
For any element a and list of elements z,

$$\begin{aligned} hsplit\ (a : z) \ = \ & (h, z_1 +\!\!+ z_2) \\ & \quad \textbf{where } (h', z_2) = hsplit\ z \\ & \qquad\qquad (h, z_1) = hsplit\ (a : h'). \end{aligned}$$

Secondly, this is how *hsplit* interacts with *mdp*:

Lemma 5.2
For any list of elements x,

$$\begin{aligned} hsplit\ (mdp\ x) \ = \ & (h, max_{\uparrow_h}\ (prefixes\ y)) \\ & \quad \textbf{where } (h, y) = hsplit\ x. \end{aligned}$$

($\uparrow_h$ is defined in Corollary 4.7.)

This lemma merely restates the left-hand side involving the input list of elements x, into a form relating to the window data structure, with its compulsory header h. Recall that *mdp* chooses an optimal prefix with respect to $\preceq$. If x is not wide enough, then both y and $mdp\ x$ evaluate to $[\,]$, and $h = x$. Otherwise, the right-hand side says that comparison with respect to $\preceq$ is carried out by comparing densities using $\leq_d^h$ (also defined in Corollary 4.7) on prefixes within the window structure.

We are now ready to transform $wbuild\ (mdp\ (a : wp\ x))$. Writing z for $wp\ x$, we calculate:

$$\begin{aligned} & wbuild\ (mdp\ (a : z)) \\ = \quad & \{ \text{ definition of } wbuild \ \} \end{aligned}$$

$$
\begin{aligned}
& (id \times drsp)\ (hsplit\ (mdp\ (a : z))) \\
= \quad & \{\ \text{property of } hsplit \text{ in Lemma 5.2, letting } (h, z') \ = \ hsplit\ (a : z)\ \} \\
& (id \times drsp)\ (h, max_{\uparrow_h}\ (prefixes\ z')) \\
= \quad & \{\ \text{letting } (h', z_2) = hsplit\ z \text{ and } (h, z_1) = hsplit\ (a : h') \text{, as in Lemma 5.1}\ \} \\
& (id \times drsp)\ (h, max_{\uparrow_h}\ (prefixes\ (z_1 +\!\!+\, z_2))) \\
= \quad & \{\ \text{since } concat \cdot drsp = id\ \} \\
& (h, drsp\ (max_{\uparrow_h}\ (prefixes\ (concat\ (drsp\ (z_1 +\!\!+\, z_2)))))) \\
= \quad & \{\ \text{Corollary 4.7}\ \} \\
& (h, drsp\ (concat\ (max_{\uparrow_{h+\!+}}\ (prefixes\ (drsp\ (z_1 +\!\!+\, z_2)))))) \\
= \quad & \{\ drsp\ (z_1 +\!\!+\, z_2) = foldr\ addl\ (drsp\ z_2)\ z_1 \text{, from the definition of } drsp \text{ in Section 4.4}\ \} \\
& (h, drsp\ (concat\ (max_{\uparrow_{h+\!+}}\ (prefixes\ (foldr\ addl\ (drsp\ z_2)\ z_1))))) \\
= \quad & \{\ \text{definition of } wbuild \text{ (see Section 4.5), letting } (h', zs_2) = wbuild\ z\ \} \\
& (h, drsp\ (concat\ (max_{\uparrow_{h+\!+}}\ (prefixes\ (foldr\ addl\ zs_2\ z_1))))) \\
= \quad & \{\ \text{definition of } wcons \text{, letting } (h, zs) = wcons\ a\ (wbuild\ z)\ \} \\
& (h, drsp\ (concat\ (max_{\uparrow_{h+\!+}}\ (prefixes\ zs)))) \\
= \quad & \{\ \text{Theorem 4.16}\ \} \\
& (h, drsp\ (concat\ (maxchop\ h\ zs))) \\
= \quad & \{\ \text{cancellation, as } zs \text{ is a DRSP (see below)}\ \} \\
& (h, maxchop\ h\ zs) \\
= \quad & \{\ \text{definition of } wmaxchop \text{ (see end of Section 4.5)}\ \} \\
& (wmaxchop \cdot wcons\ a \cdot wbuild)\ z.
\end{aligned}
$$

The cancellation in the penultimate step is valid because of the following: *wbuild* builds a DRSP, and *wcons* and *maxchop* return DRSPs when given a DRSP as input. Then, as DRSPs are unique, applying $drsp \cdot concat$ to a DRSP has no effect.

In summary, we have shown that

$$wbuild \cdot mdp \cdot (a :) = wmaxchop \cdot wcons\ a \cdot wbuild. \tag{26}$$

We resume the calculation of the inductive case for *mwp*:

$$
\begin{aligned}
& mwp\ (a : x) \\
= \quad & \{\ \text{by previous calculation and (26)}\ \} \\
& (wflatten\ u \uparrow_{\preceq} ms\ x, u) \\
& \textbf{where } u = wmaxchop\ (wcons\ a\ (wbuild\ (wp\ x))) \\
= \quad & \{\ \text{adding variables } m \text{ and } w\ \} \\
& (wflatten\ u \uparrow_{\preceq} m, u) \\
& \textbf{where } m = ms\ x \\
& \qquad\quad w = wbuild\ (wp\ x) \\
& \qquad\quad u = wmaxchop\ (wcons\ a\ w)
\end{aligned}
$$

$$
\begin{array}{l}
= \quad \{ \text{ definition of } \mathit{mwp} \ \} \\
\quad (\mathit{wflatten}\ u \uparrow_{\preceq} m, u) \\
\quad \textbf{where}\ (m, w) = \mathit{mwp}\ x \\
\qquad\qquad u = \mathit{wmaxchop}\ (\mathit{wcons}\ a\ w).
\end{array}
$$

We have thus constructed an inductive alternative definition for *mwp*:

$$
\begin{array}{ll}
\mathit{mwp} & ::\ [\mathit{Elem}] \to ([\mathit{Elem}], \mathit{Window}) \\
\mathit{mwp}\ [\,] & =\ ([\,], ([\,], [\,])) \\
\mathit{mwp}\ (a : x) & = (\mathit{wflatten}\ u \uparrow_{\preceq} m, u) \\
 & \textbf{where}\ (m, w) = \mathit{mwp}\ x \\
 & \qquad\quad u = \mathit{wmaxchop}\ (\mathit{wcons}\ a\ w).
\end{array}
$$

As a maximally dense segment is obtained from $\mathit{fst} \cdot \mathit{mwp}$, we have arrived at our algorithm, which is summarised in the following section.

5.2 Data Refinement and Performance Analysis

In this section we perform some final data structure refinement, to ensure that the final algorithm runs in linear time (and space). For the reader's convenience, Figure 8 summarises the derived program, which includes an implemention of the function *hsplit*. In this code, the type *Window* is give more abstractly as

$$\textbf{type}\ \mathit{Window}\ =\ (\mathit{Header}\ \mathit{Elem}, \mathit{Parts}\ (\mathit{Seg}\ \mathit{Elem})).$$

The types *Header*, *Parts*, and *Seg* are respectively the datatypes with which we implement the header, the partition, and each segment in the partition. These datatypes can be thought of as instances of a *List* class, with polymorphic operators such as $[\,]$, $(:)$, and $(+\!\!+)$, and we use these operators in the code for clarity.

Lists of elements of type *Header*, *Parts*, or *Seg* all need to have their density and width computable in constant time. This can easily be done by pairing the lists with their current values for weight, length, width, etc., and is considered a separate issue from the implementation details that follow.

The data structure used for the *Header* datatype depends on the implementation of the *hsplit* function, as within the algorithm, the header is only altered using *hsplit*. One possible implementation makes use of an auxiliary function *split*, which repeatedly removes the rightmost element of the header until it does not exceed the width limit, while retaining as much width as possible. Since each element is initially added to the header on the left and later removed from the right, at most once, *hsplit* runs in linear time if addition on the left and removal from the right are both amortised constant time operations. This can be done by implementing *Header* as a simple queue using two lists, e.g. see Okasaki (1999).

For the *Parts* and *Seg* datatypes, we need to examine the operations on the DRSP structure of the window. Elements are added one-by-one on the left by *addl*, which just launches *prepend*. The function *prepend* makes an indefinite number of recursive calls, and in each call two segments are joined. However, notice that segments in the DRSP are all non-empty, and are never split once joined. Therefore, given an input of length n, there can be at most $O(n)$ joins in total. The function *prepend* thus runs in linear time, provided

```
ms :: [Elem] → [Elem]
ms = fst · mwp

type Window = (Header Elem, Parts (Seg Elem))

mwp         :: [Elem] → ([Elem], Window)
mwp []      = ([], ([], []))
mwp (a : x) = (wflatten u ↑≼ m, u)
       where (m, w) = mwp x
             u = wmaxchop (wcons a w)
```

```
hsplit    :: Header Elem → (Header Elem, [Elem])
hsplit x  = split (x, [])

split                :: (Header Elem, [Elem]) → (Header Elem, [Elem])
split ([], y)        = ([], y)
split (x ++ [a], y)  = if width x < L then (x ++ [a], y)
                       else split (x, a : y)
```

```
wcons            :: Elem → Window → Window
wcons a (h, xs)  = (h′, foldr addl xs x)
            where (h′, x) = hsplit (a : h)

addl      :: Elem → Parts (Seg Elem) → Parts (Seg Elem)
addl a xs = prepend [a] xs

prepend            :: Seg Elem → Parts (Seg Elem) → Parts (Seg Elem)
prepend x []       = [x]
prepend x (y : xs) = if x ≤d y then prepend (x ++ y) xs
                     else x : y : xs
```

```
wmaxchop          :: Window → Window
wmaxchop (h, xs)  = (h, maxchop h xs)

maxchop                :: Header Elem → Parts (Seg Elem) → Parts (Seg Elem)
maxchop h []           = []
maxchop h (xs ++ [x])  = if (h ++ concat xs) <d x then xs ++ [x]
                         else maxchop h xs
```

Fig. 8. The derived algorithm.

that *Parts* allows addition to and removal from the left in amortised constant time, and that *Seg* supports list concatenation in (amortised) constant time. The requirement on *Seg* is easy to fulfill: one may simply use a join list: **data** $Seg\ a = Single\ a \mid Join\ (Seg\ a)\ (Seg\ a)$.

The removal of elements from the window is carried out by the *maxchop* function, which removes segments in *Parts* from the right. Therefore we need *Parts* to support amortised constant-time removal from both ends. A number of data structures support such operations, for example, Banker's dequeues (Okasaki, 1999), or 2-3 finger trees (Hinze & Paterson, 2006).

We tested our Haskell implementation, compiled using the Glasgow Haskell Compiler (GHC) version 7.10.1 and run on a Intel Xeon Quad-Core E5620 PC running OS X, at 2.40GHz. Sample timing measurements are presented in Figure 9.

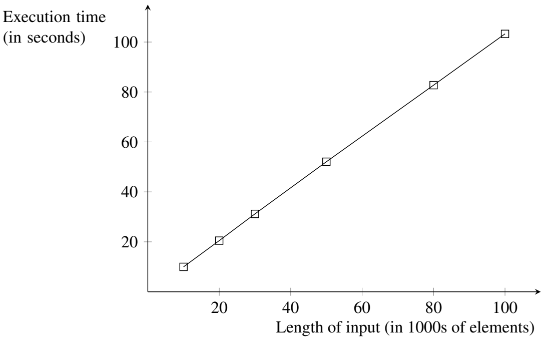

Fig. 9. Measurements of mean (over 20 lists) running time for the MDS implementation. Sizes were randomly chosen from the range $[1..20]$, weights from the range $[-100,100]$, and $L = 400$.

Timing measurements were also carried out with different values for the lower bound L, and different ranges and probability distributions for the numerical input. In all cases, the resulting timings were very close to those in Figure 9 (too similar to distinguish, if plotted on the same diagram). Typically, the maximally-dense segments produced are only a little wider than the width constraint, and in order to give the window data structure a more thorough test, we also created special cases designed to produce much wider solutions. Again, the timings were very similar to those illustrated above.

Thus the empirical evidence is consistent thoroughout with our analysis of a linear-time efficiency for the algorithm.

6 Conclusions

We have derived a linear-time algorithm for solving the generalised segment density problem with a lower bound on segment width. The algorithm scans through the input list using a sliding window, which is split into a header and a partition of right-skew segments with decreasing densities (DRSP), whose properties we exploit to make the linear-time processing possible. While the program itself barely occupies one page, its proof is anything but simple, involving the discovery of intricate properties, and our complete proof of the algorithm uses approximately 3500 lines of Agda code.

Two instances of the density segment problem were presented. While the maximum density segment (MDS) problem has a long history, we believe that the maximum mean segment (MMS) problem is new. These are similar but neither is a generalisation of the other: in MDS, the density function is fundamentally linked to the segment width, but this is not the case for the MMS problem. This shows that, for the densest segment problem, width can be separate from density, allowing the algorithm to analyse data such as blood glucose measurements.

The development of our solution has not been straightforward, and we faced a number of design decisions from the beginning. Should we model the problem functionally, relationally, or use total relations? As there were technical complications involving relations and zygomorphisms (see the technical aside in Section 3.4), a functional approach was strongly indicated. Another decision concerned the maximum operators $\upharpoonleft$ or $\upharpoonright$, when deciding which segment to return in case of a tie: which to use, and did it matter? It turns out that the choice does matter — choosing $\upharpoonright$ would have meant that the algorithm would fail to meet that specification. Furthermore, the algorithm might not yield a result when there is no segment within the width bound, but using a *Maybe* type made the reasoning rather cumbersome. Using an extended ordering $\preceq$ allows a much cleaner formulation.

While we had an informal understanding of why this algorithm is correct, formally writing down the properties that make it so turned out to be surprisingly tricky. The correctness of the "outer" algorithm (Section 3.6), treated casually in all previous works, took us a considerable amount of time to formalise. We attempted to come up with a more declarative specification of the prefix returned by *wp* (for example, we guessed that it is the shortest prefix satisfying certain properties), but none of those specifications were correct. Various possibilities were tried before we reached Lemma 3.7 and its supporting definitions. Afterwards, the proof quickly followed. It is often the case that finding the right thing to prove is harder than producing the proof. It was surprising that in Lemma 3.7 we could prove an equality, rather than merely that the two sides yield segments having the same density. In fact, we needed an equality for the inductive proof to work.

All this hard work was not spent in vain. We re-proved the uniqueness of the DRSP, and presented various properties about it, including how it can be constructed. This fills in more properties about the DRSP than given in previous papers.

We initially set out to solve a more general version of the MDS problem, with an additional upper bound on segment width, making the problem even more intricate. As mentioned in Section 1, an algorithm with a wrong time analysis has been published before. Even the algorithms of Chung & Lu (2004) and Goldwasser *et al.* (2005) are not entirely correct: they both fail for a boundary case, specifically when there is no segment in the window whose width is within the bounds. The former algorithm could potentially return a invalid result, for which there is an easy fix (Chung, 2010), while the latter loops and it is harder to see whether it is fixable.

We have developed data structures that are used in the sliding window to produce densest segments with an upper bound on their width, and we have some preliminary results on correctness proofs, which will have to be deferred to another paper.

We believe that the difficulty in developing correct linear-time algorithms is partly due to the complicated nature of the MDS problem, and partly due to the absence of a rigorous approach to program construction. The imperative algorithms of Chung & Lu and Goldwasser both maintain invariants that are neither explicitly stated nor easy to reconstruct. The invariants rely on states stored in static variables surviving between subroutine calls, which makes reasoning about them extremely hard. In addition, their liberal use of array indices obscures some beautiful structural properties of segment densities. The MDS problem provides a useful case study of how formal development techniques can help with constructing correct programs.

Acknowledgments

We would like to thank Richard Bird for stimulating discussions of the MDS problem. Many thanks are also due to the anonymous referees, whose detailed comments helped to improve our presentation.

References

Bird, Richard S. (1987). An introduction to the theory of lists. *Page 342 of:* Broy, Manfred (ed), *Logic of Programming and Calculi of Discrete Design*. NATO ASI Series F, no. 36. Springer-Verlag.

Bird, Richard S., & de Moor, Oege. (1997). *The Algebra of Programming*. Prentice Hall.

Chung, Kai-Min. (2010). Personal communication.

Chung, Kai-Min, & Lu, Hsueh-I. (2003). An optimal algorithm for the maximum-density segment problem. *Pages 136–147 of: Annual European Symposium on Algorithms*. Lecture Notes in Computer Science, no. 2832.

Chung, Kai-Min, & Lu, Hsueh-I. (2004). An optimal algorithm for the maximum-density segment problem. *SIAM Journal on Computing*, **34**(2), 373–387.

Curtis, Sharon, & Mu, Shin-Cheng. (2014). *Calculating a linear-time solution to the densest segment problem (paper and supplement materials)*. `http://www.iis.sinica.edu.tw/~scm/2014/mds/`.

Goldwasser, Michael H., Kao, Ming-Yang, & Lu, Hsueh-I. (2002). Fast algorithms for finding maximum-density segments of a sequence with applications to bioinformatics. *Pages 157–171 of: Proceedings of the 2nd Workshop on Algorithms in Bioinformatics (WABI 2002)*.

Goldwasser, Michael H., Kao, Ming-Yang, & Lu, Hsueh-I. (2005). Linear-time algorithms for computing maximum-density sequence segments with bioinformatics applications. *Journal of Computer and System Sciences*, **70**(2), 128–144.

Han, Leng, & Zhao, Zhongming. (2009). CpG islands or CpG clusters: how to identify functional gc-rich regions in a genome? *BMC Bioinformatics*, **10**(65).

Hinze, Ralf, & Paterson, Ross. (2006). Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, **16**(2), 197–217.

Huang, Xiaoqui. (1994). An algorithm for identifying regions of a DNA sequence that satisfy a content requirement. *Computer Applications in the Biosciences*, **10**(3), 219–225.

Jeuring, Johan T. (1993). *Theories for Algorithm Calculation*. Ph.D. thesis, Utrecht University.

Jeuring, Johan T., & Meertens, Lambert. (1989). The least-effort cabinet formation. *The Squiggolist*, **1**(2), 12–16.

Kaldewaij, Anne. (1990). *Programming: the Derivation of Algorithms*. Prentice Hall.

Lin, Yaw-Ling, Jiang, Tao, & Chao, Kun-Mao. (2002). Efficient algorithms for locating the length-constrained heaviest segments, with applications to biomolecular sequence analysis. *Pages 459–470 of: Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science, no. 2420.

Malcolm, Grant R. (1990). *Algebraic Data Types and Program Transformation*. Ph.D. thesis, Groningen University, The Netherlands.

Mu, Shin-Cheng, Ko, Hsiang-Shang, & Jansson, Patrik. (2009). Algebra of programming in Agda: dependent types for relational program derivation. *Journal of Functional Programming*, **19**(5), 545–579.

Norell, Ulf. (2007). *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. thesis, Chalmers University of Technology.

Okasaki, Chris. (1999). *Purely Functional Data Structures*. Cambridge University Press.

Rem, Martin. (1988). Small programming exercises 20. *Science of Computer Programming*, **10**(1), 99–105.

Swierstra, S.D., & de Moor, O. (1993). Virtual data structures. *Pages 355–371 of:* Bernhard Möller, Helmut Partsch, Steve Schumann (ed), *IFIP TC2/WG2.1 State-of-the-Art Report on Formal Program Development,*. Lecture Notes in Computer Science. Springer.

Van Den Eijnde, J.P.H.W. (1990). Left-bottom and right-top segments. *Science of Computer Programming*, **15**(1), 79–94.

Zantema, Hans. (1992). Longest segment problems. *Science of Computer Programming*, **18**(1), 39–66.